03 | 档案类：怎么精简地表达不可变数据？

2021-11-19 范学雷

《深入剖析Java新特性》 课程介绍 >

讲述：范学雷
时长 22:25 大小 20.54M

你好，我是范学雷。今天，我们聊一聊 Java 的档案类。

档案类这个特性，首先在 JDK 14 中以预览版的形式发布。在 JDK 15 中，改进的档案类再次以预览版的形式发布。最后，档案类在 JDK 16 正式发布。

那么，什么是档案类呢？档案类的英文，使用的词汇是“record”。官方的说法，Java 档案类是用来表示不可变数据的透明载体。这样的表述，有两个关键词，一个是不可变的数据，另一个是透明的载体。

该怎么理解“不可变的数据”和“透明的载体”呢？我们还是通过案例和代码，一步一步地来拆解、理解这些概念。

阅读案例

在面向对象的编程语言中，研究表示形状的类是一个常用的教学案例。今天的评审案例，我们从形状的子类圆形开始，来看一看面向对象编程实践中，这个类的设计和演化。

下面的这段代码，就是一个简单的、典型的圆形类的定义。这个抽象类的名字是 **Circle**。它有一个私有的变量 **radius**，用来表示圆的半径。有一个构造方法，用来生成圆形的实例。有一个设置半径的方法 **setRadius**，一个读取半径的方法 **getRadius**。还有一个重载的方法 **getArea**，用来计算圆形的面积。

复制代码

```
1  package co.ivi.jus.record.former;
2
3  public final class Circle implements Shape {
4      private double radius;
5
6      public Circle(double radius) {
7          this.radius = radius;
8      }
9
10     @Override
11     public double getArea() {
12         return Math.PI * radius * radius;
13     }
14
15     public double getRadius() {
16         return radius;
17     }
18
19     public void setRadius(double radius) {
20         this.radius = radius;
21     }
22 }
```

这个圆形类之所以典型，是因为它交代了面向对象设计的关键思想，包括面向对象编程的三大支柱性原则：封装、继承和多态。

封装的原则是隐藏具体实现细节，实现的修改不会影响接口的使用。**Circle** 类中，表示半径的变量被定义成私有的变量。我们可以改变半径这个变量的名字，或者不使用半径而是使用直径来表示圆形。这样的实现细节的变化，并不会影响公开方法的调用。

由于需要隐藏内部实现细节，所以需要设计公开接口来访问类的相关特征，比如例子中的圆形的半径。所以上面的例子中，设置半径的方法 **setRadius** 和读取半径的方法 **getRadius**，就显得显而易见，并且顺理成章。在面向对象编程的教科书里，以及 Java 的标准类库里，我们可以看到很多类似的设计。

可是，这样的设计有哪些严重的缺陷呢？花点时间想想你能找到的问题，然后我们接下来再继续分析。

案例分析

上面这个例子，最重要的问题，就是它的接口不是多线程安全的。如果在一个多线程的环境中，有些线程调用了 **setRadius** 方法，有些线程调用 **getRadius** 方法，这些调用的最终结果是难以预料的。这也就是我们常说的多线程安全问题。

在现代计算机架构下，大多数的应用需要多线程的环境。所以，我们通常需要考虑多线程安全的问题。 该怎么解决上面例子中的多线程安全问题呢？如果上述例子的实现源代码不能更改，那么就需要在调用这些接口的程序中，增加线程同步的措施。

复制代码

```
synchronized (circleObject) {
    double radius = circleObject.getRadius();
    // do something with the radius.
}
```

遗憾的是，在调用层面解决线程同步问题的办法，并不总是显而易见的。不论多么资深的程序员，都有可能疏漏、忘记或者没有正确地解决好线程同步的问题。

所以，通常地，为了更皮实的接口设计，在接口规范设计的时候，就应该考虑解决掉线程同步的问题。比如说，我们可以把上面案例中的代码改成线程安全的代码。对于 **Circle** 类，只需要把它的公开方法都设置成同步方法，那么这个类就是多线程安全的了。具体的实现，请参考下面的代码。

复制代码

```
package co.ivi.jus.record.former;

public final class Circle implements Shape {
    private double radius;
```

```
5
6       public Circle(double radius) {
7           this.radius = radius;
8       }
9
10      @Override
11      public synchronized double getArea() {
12          return Math.PI * radius * radius;
13      }
14
15      public synchronized double getRadius() {
16          return radius;
17      }
18
19      public synchronized void setRadius(double radius) {
20          this.radius = radius;
21      }
22  }
```

可是，线程同步并不是免费的午餐。代价有多大呢？我做了一个简单的性能基准测试，哪怕最简单的同步，比如上面代码里同步的 **getRadius** 方法，它的吞吐量损失也有十数倍。这相当于说，如果没有同步的应用需要一台机器支持的话，加了同步的应用就需要十多台机器来支撑相同的业务量。

这样的代价就有点大了，我们需要寻找更好的办法来解决多线程安全的问题。最有效的办法，就是在接口设计的时候，争取做到即使不使用线程同步，也能做到多线程安全。这说起来还是有点难以理解的，我们还是来看看代码吧。

下面的代码，是一个修改过的 **Circle** 类实现。在这个实现里，圆形的对象一旦实例化，就不能再修改它的半径了。相应地，我们删除了设置半径的方法。也就是说，这个对象是一个只读的对象，不支持修改。通常地，我们称这样的对象为不可变对象。

复制代码

```
1  package co.ivi.jus.record.immute;
2
3  public final class Circle implements Shape {
4      public final double radius;
5
6      public Circle(double radius) {
7          this.radius = radius;
8      }
9
10     @Override
```

```
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

对于只读的圆形类的设计，我们可以看到两个好处。

第一个好处，就是天生的多线程安全。因为这个类的对象，一旦实例化就不能再修改，所以即便在多线程环境下使用，也不需要同步。而不可变对象所承载的数据，比如上面例子中圆形的半径，就是我们前面所说的不可变的数据。这个不可变，是有一个界定范围的。这个界定范围，就是它所在对象的生命周期。如果跳出了对象的生命周期，我们可以重新生成新对象，从而实现数据的变化。

第二个好处，就是简化的代码。只读对象的设计，使得我们可以重新考虑代码的设计，这是代码简化的来源。你可能已经注意到了，在这个实现里，我们还删除了读取半径的方法。取而代之的，是公开的半径这个变量。这就是一个最直接的简化。

应用程序可以直接读取这个变量，而不是通过一个类似于 getRadius 的方法。由于半径这个变量被声明为 final 变量，所以它只可以被读取，不能被修改。这并没有破坏对象的只读性。

不过，乍看之下，这样的设计似乎破坏了面向对象编程的封装原则。公开半径变量 radius，相当于公开的实现细节。如果我们改变主意，想使用直径来表示一个圆形，那么实现的修改就会显得很丑陋。

可是，如果我们认真思考一下几个简单的问题，对于封装的顾虑可能就降低很多了。比如说，使用直径来表示一个圆，这是一个真实的需求吗？ 这是一个必需的表达方式吗？未来的圆，会不会变得没法使用半径来表达？其实不是的，未来的圆，还是可以用半径来表达的。使用其他的办法，比如直径，来表达一个圆，其实并没有必要。

所以，公开半径这个只读变量，并没有带来违反封装原则的实质性后果。而且，从另外一个角度来看，我们可以把读取这个只读变量的操作，看成是等价的读取方法的调用。不过，虽然很多人，包括我自己，倾向于这样解读，但是这总归是一个有争议的形式。

进一步的简化

还有没有进一步简化的空间呢？我们再来看看不可变的正方形 **Square** 类的设计。具体的实现，请参考下面的代码。

复制代码

```
package co.ivi.jus.record.immute;

public final class Square implements Shape {
    public final double side;

    public Square(double side) {
        this.side = side;
    }

    @Override
    public double area() {
        return side * side;
    }
}
```

如果比较一下不可变的圆形 Circle 类和正方形 Square 类的源代码，你有没有发现这两个类的代码有惊人的相似点？

第一个相似的地方，就是使用公开的只读变量（使用 final 修饰符来声明只读变量）。Circle 类的变量 **radius**，和 Square 类的变量 **side**，都是公开的只读的变量。这样的声明，是为了公开变量的只读性。

第二个相似的地方，就是公开的只读变量，需要在构造方法中赋值，而且只在构造方法中赋值，且这样的构造方法还是公开的方法。Circle 类的构造方法给 **radius** 变量赋值，Square 类的构造方法给 **side** 变量赋值。这样的构造方法，解决了对象的初始化问题。

第三个相似的地方，就是没有了读取的方法；公开的只读变量，替换了掉了公开的读取方法。这样的变化，使得代码量总体变少了。

这么多相似的地方，相似的代码，能不能进一步地简化呢？我知道，你可能已经开始思考这样的问题了。

对于这个问题，Java 的答案，就是使用档案类。

怎么声明档案类

我们前面说过，Java 档案类是用来表示不可变数据的透明载体。那么，怎么使用档案类来表示不可变数据呢？

我们还是一起先来看看代码吧。咱们试着把上面不可变的圆形 Circle 普通的类改成档案类，来感受下档案类到底是什么模样的。

复制代码

```
package co.ivi.jus.record.modern;

public record Circle(double radius) implements Shape {
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

看到这样的代码，是不是有点出乎意料？你可以对比一下不可变的 Circle 类的代码，感受一下这两者之间的差异。

首先，最常见的 class 关键字不见了，取而代之的是 record 关键字。record 关键字是 class 关键字的一种特殊表现形式，用来标识档案类。record 关键字可以使用和 class 关键字差不多一样的类修饰符（比如 public、static 等；但是也有一些例外，我们后面再说）。

然后，类标识符 Circle 后面，有用小括号括起来的参数。类标识符和参数一起看，就像是一个构造方法。事实上，这样的表现方式，的确可以看成是构造方法。而且，这种形式，还就是当作构造方法使用的。比如下面的代码，就是使用构造方法的形式来生成 Circle 档案类实例的。

复制代码

```
Circle circle = new Circle(10.0);
```

最后，在大括号里，也就是档案类的实现代码里，变量的声明没有了，构造方法也没有了。前面我们已经知道怎么生成一个档案类实例了，但还有一个问题是，我们能读取这个

圆形档案类的半径吗？

其实，类标识符声明后面的小括号里的参数，就是等价的不可变变量。在档案类里，这样的不可变变量是私有的变量，我们不可以直接使用它们。但是我们可以通过等价的方法来调用它们。变量的标识符就是等价方法的标识符。比如下面的代码，就是一个读取上面圆形档案类半径的代码。

复制代码

```
1 double radius = circle.radius();
```

是的，在档案类里，方法调用的形式又回来了。我们前面讨论过打破封装原则的顾虑，你可能还是没有足够的信心去接受不完整的封装形式。那么现在，档案类的调用形式依然保持着良好的封装形式。打破封装原则的顾虑也就不复存在了。

需要注意的是，由于档案类表示的是不可变数据，除了构造方法之外，并没有给不可变变量赋值的方法。

意料之外的改进

上面，通过传统 Circle 类和档案 Circle 类代码的对比，我们可以感受到档案类在简化代码、提高生产力方面的努力。如果说，上面这些简化，还在我的预料之内的话；下面的简化，我刚看到的时候，是很惊喜的：“哇，这真是太奇妙了！”

我们还是通过代码来体验一下这种感受。如果我们生成两个半径为 10 厘米的圆形的实例，这两个实例是相等的吗？下面的代码，就是用来验证我们猜想的。你可以试着运行一下，看看和你猜想的结果是不是一样的。

复制代码

```
1  package co.ivi.jus.record;
2
3  import co.ivi.jus.record.immute.Circle;
4
5  public class ImmuteUseCases {
6      public static void main(String[] args) {
7          Circle c1 = new Circle(10.0);
8          Circle c2 = new Circle(10.0);
9
10         System.out.println("Equals? " + c1.equals(c2));
```

```
    }
}
```

上面的代码里，使用了我们开篇案例分析中的传统 Circle 类。运行结果告诉我们，两个半径为 10 厘米的圆形的实例，并不是相等的实例。我想这应该在你的预料之内。

如果需要比较两个实例是不是相等，我们需要重载 equals 方法和 hashCode 方法。如果需要把实例转换成肉眼可以阅读的信息，我们需要重载 toString 方法。我们上面案例分析的代码中，这些方法都没有重载，因此对应的操作结果也是不可预测的。

当然，如果没有遗忘，我们可以添加这三个方法的重载实现。然而，这三个方法的重载，尤其是 equals 方法和 hashCode 方法的重载实现，一直是代码安全的重灾区。即便是经验丰富的程序员，也可能忘记重载这三个方法；就算没有遗忘，equals 方法和 hashCode 方法也可能没有正确实现，从而带来各种各样的问题。这实在难以让人满意，但是一直以来，我们也没有更好的办法。

档案类会不一样吗？

我们再来看看使用档案类的代码，结果会不会不一样呢？ 下面的这段代码，Circle 的实现使用的是档案类。这段代码运行的结果告诉我们，两个半径为 10 厘米的圆形的档案类实例，是相等的实例。

复制代码

```
package co.ivi.jus.record;

import co.ivi.jus.record.modern.Circle;

public class ModernUseCases {
    public static void main(String[] args) {
        Circle c1 = new Circle(10.0);
        Circle c2 = new Circle(10.0);

        System.out.println("Equals? " + c1.equals(c2));
    }
}
```

看到这里，你是不是感觉到：哇！ 这真的是太棒了！我们并没有重载这三个方法，它们居然可以使用。

为什么会这样呢？

这是因为，档案类内置了缺省的 equals 方法、hashCode 方法以及 toString 方法的实现。一般情况下，我们就再也不用担心这三个方法的重载问题了。这不仅减少了代码数量，提高了编码的效率；还减少了编码错误，提高了产品的质量。

不可变的数据

讨论到这里，我们可以回头再看看 Java 档案类的定义了：Java 档案类是用来表示不可变数据的透明载体。“不可变的数据”和“透明的载体”是两个最重要的关键词。

我们前面讨论了不可变的数据。如果一个 Java 类一旦实例化就不能再修改，那么用它表述的数据就是不可变数据。Java 档案类就是表述不可变数据的。为了强化“不可变”这一原则，避免面向对象设计的陷阱，Java 档案类还做了以下的限制：

1. Java 档案类不支持扩展子句，用户不能定制它的父类。隐含的，它的父类是 java.lang.Record。父类不能定制，也就意味着我们不能通过修改父类来影响 Java 档案的行为。
2. Java 档案类是个终极（final）类，不支持子类，也不能是抽象类。没有子类，也就意味着我们不能通过修改子类来改变 Java 档案的行为。
3. Java 档案类声明的变量是不可变的变量。这就是我们前面反复强调的，一旦实例化就不能再修改的关键所在。
4. Java 档案类不能声明可变的变量，也不能支持实例初始化的方法。这就保证了，我们只能使用档案类形式的构造方法，避免额外的初始化对可变性的影响。
5. Java 档案类不能声明本地（native）方法。如果允许了本地方法，也就意味着打开了修改不可变变量的后门。

通常地，我们把 Java 档案类看成是一种特殊形式的 Java 类。除了上述的限制，Java 档案类和普通类的用法是一样的。

透明的载体

好了，聊完“不可变的数据”，接下来该聊聊“透明的载体”了。

陆陆续续地，我们在前面提到过，档案类内置了下面的这些方法缺省实现：

构造方法

equals 方法

hashCode 方法

toString 方法

不可变数据的读取方法

如果你注意到的话，我们使用了“缺省”这样的字眼。换一种说法，我们可以使用缺省的实现，也可以替换掉缺省的实现。下面的代码，就是我们试图替换掉缺省实现的尝试。请注意，除了构造方法，其他的替换方法都可以使用 **Override** 注解来标注（如果你读过《代码精进之路》，你就会倾向于总是使用 **Override** 注解的）。

复制代码

```
package co.ivi.jus.record.explicit;

import java.util.Objects;

public record Circle(double radius) implements Shape {
    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }

        if (o instanceof Circle other) {
            return other.radius == this.radius;
        }

        return false;
    }
```

```
27
28      @Override
29      public int hashCode() {
30          return Objects.hash(radius);
31      }
32
33      @Override
34      public String toString() {
35          return String.format("Circle[radius=%f]", radius);
36      }
37
38      @Override
39      public double radius() {
40          return this.radius;
41      }
42  }
```

到这里，你应该明白了“透明的载体”的意思了。透明载体的意思，通俗地说，就是档案类承载有缺省实现的方法，这些方法可以直接使用，也可以替换掉。

不过，像上面这样的替换，除了徒增烦恼，是没有实际意义的。那我们什么时候需要替换掉缺省实现呢？

重载构造方法

最常见的替换，是要在构造方法里对档案类声明的变量添加必要的检查。比如说，我们现实生活中看到的各种各样的圆形，它的半径都不会是负数。如果在这样的场景里来讨论圆形，那么表示圆形的类的半径就不应该是负数。

你应该已经意识到了，我们上面的代码，在实例化的时候，都没有检查半径的数值，包括档案类缺省的构造方法。那么这时候，我们就要替换掉缺省的构造方法。下面的代码，就是一种替换的方法。如果，构造实例的时候，半径的数值为负，构造就会抛出运行时异常**IllegalArgumentException**。

复制代码

```
1  package co.ivi.jus.record.improved;
2
3  public record Circle(double radius) implements Shape {
4      public Circle {
5          if (radius < 0) {
6              throw new IllegalArgumentException(
7                  "The radius of a circle cannot be negative [" + radius + "]");
```

```
        }
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

如果你阅读了上面的代码，应该已经注意到了一点不太常规的形式。构造方法的声明没有参数，也没有给实例变量赋值的语句。这并不是说，构造方法就没有参数，或者实例变量不需要赋值。实际上，为了简化代码，Java 编译的时候，已经替我们把这些东西加上去了。所以，不论哪一种编码形式，构造方法的调用都是没有区别的。

在上一个例子中，我们已经看到了构造方法的常规形式。在下面这张表里，我列出了两种构造方法形式上的差异，你可以看看它们的差异。

声明的形式	代码	注释

常规的形式	public Circle(double radius) {
	  if (radius < 0) {
	  // snipperd
	  }
	
	  this.radius = radius;
	}
		1、构造方法的声明包含参数。
		2、有给实例的变量赋值的语句。

简化的形式	public Circle {
	  if (radius < 0) {
	  // snipped
	  }
	}
		1、构造方法的声明不包含参数，也没有小括号。
		2、没有给实例变量赋值的语句。

极客时间

重载 equals 方法

还有一类常见的替换，如果缺省的 equals 方法或者 hashCode 方法不能正常工作或者存在安全的问题，就需要替换掉缺省的方法。

如果声明的不可变变量没有重载 equals 方法和 hashCode 方法，那么这个档案类的 equals 方法和 hashCode 方法的行为就可能不是可以预测的。比如，如果不可变的变量是一个数组，通过下面的例子，我们来看看它的 equals 方法能不能正常工作。

复制代码

```
jshell> record Password(byte[] password) {};
|  modified record Password

jshell> Password pA = new Password("123456".getBytes());
pA ==> Password[password=[B@2ef1e4fa]

jshell> Password pB = new Password("123456".getBytes());
pB ==> Password[password=[B@b81eda8]

jshell> pA.equals(pB);
$16 ==> false
```

这个例子里，我们设计了一个口令的档案类，其中的口令使用字节数组来存放。我们使用同样的口令，生成了两个不同的实例。然后，我们调用 equals 方法，来比较这两个实例。

运算的结果显示，这两个实例并不相等。这不是我们期望的结果。其中的原因，就是因为数组这个变量的 equals 方法并不能正常工作（或者换个说法，数组变量没有重载 equals 方法）。

如果把变量的类型换成重载了 equals 方法的字符串 String，我们就能看到预期的结果了。

复制代码

```
jshell> record Password(String password) {};
|  created record Password

jshell> Password pA = new Password("123456");
pA ==> Password[password=123456]

jshell> Password pB = new Password("123456");
pB ==> Password[password=123456]

jshell> pA.equals(pB);
$5 ==> true
```

一般情况下，equals 方法和 hashCode 方法是成双成对的，实现逻辑上需要匹配。所以，当我们重载 equals 方法的时候，一般也需要重载 hashCode 方法；反之亦然。

不推荐的重载

为了更个性化的显示，我们有时候也需要重载 toString 方法。但是，我们通常不建议重载不可变数据的读取方法。因为，这样的重载往往意味着需要变更缺省的不可变数值，从而打破实例的状态，进而造成许多无法预料的、让人费解的后果。

比如说，我们设想定义一个数，如果是负值的话，我们希望读取的是它的相反数。下面的例子，就是一个味道很坏的示范。

复制代码

```
jshell> record Number(int x) {
   ...>     public int x() {
   ...>         return x > 0 ? x : (-1) * x;
   ...>     }
   ...> }
|  created record Number

jshell> Number n = new Number(-1);
n ==> Number[x=-1]

jshell> n.x();
$9 ==> 1

jshell> Number m = new Number(n.x());
m ==> Number[x=1]

jshell> m.equals(n);
$11 ==> false
```

在这个例子里，我们重载了读取的方法。如果一个数是负数，重载的读取就返回它的相反数。读取出来的数据，并不是实例化的时候赋于的数据。这让代码变得难以理解，很容易出错。

更严重的问题是，这样的重载不再能够支持实例的拷贝。比如说，我们把实例 n 拷贝到另一个实例 m。这两个实例按照道理来说应该相等。而由于重载了读取的方法，实际的结果，这两个实例是不相等的。这样的结果，也可能会使代码容易出错，而且难以调试。

总结

好，今天就到这里，我来做个小结。从前面的讨论中，我们了解到，Java 档案类是用来表示不可变数据的透明载体，用来简化不可变数据的表达，提高编码效率，降低编码错误。同时，我们也讨论了使用档案类的几个容易忽略的陷阱。

在我们日常的接口设计和编码实践中，为了最大化的性能，我们应该优先考虑使用不可变的对象（数据）；如果一个类是用来表述不可变的对象（数据），我们应该优先使用 Java 档案类。

如果要丰富你的代码评审清单，有了封闭类后，你可以加入下面这一条：

> 一个类，如果是用来表述不可变的数据，能不能使用 Java 档案类？

另外，通过今天的讨论，我拎出几个技术要点，这些都可能在你们面试中出现哦，通过学习，你应该能够：

知道 Java 支持档案类，并且能够有意识地使用档案类，提高编码效率，降低编码错误；

- 面试问题：你知道档案类吗？会不会使用它？

了解档案类的原理和它要解决的问题，知道使用不可变的对象优势；

- 面试问题：什么情况下可以使用档案类，什么情况下不能使用档案类？

了解档案类的缺省方法，掌握缺省方法的好处和不足，知道什么时候要重载这些方法。

- 面试问题：使用档案类应该注意什么问题？

如果你能够有意识地使用不可变的对象以及档案类，并且有能力规避掉其中的陷阱，你应该能够大幅度提高编码的效率和质量。毫无疑问，在面试的时候，这也是一个能够让你脱颖而出的知识点。

思考题

在重载 equals 方法这一小节里，我们讨论了数组类型的不可变数据。我们已经知道了，这样的数据类型，需要重载 equals 方法和 hashCode 方法。其实，toString() 的方法也需要重载。今天的思考题，就是请你实现这些方法的重载。

方便起见，我们假设这个数组是字节数组，用来表示社会保障号。我们都知道，社会保障号是高度敏感的信息，不能被泄漏，也不能被盗取。你来想一想，有哪些方法需要重载？为什么？代码看起来是什么样子的？有难以克服的困难吗？

我开个头，写一个空白的档案类，你来把你想添加的代码补齐。

复制代码

```
record SocialSecurityNumber(byte[] ssn) {
  // Here is your code.
}
```

欢迎你在留言区留言、讨论，分享你的阅读体验以及对这些问题的思考。

注：本文使用的完整的代码可以从GitHub下载，你可以通过修改GitHub上review template代码，完成这次的思考题。如果你想要分享你的修改或者想听听评审的意见，请提交一个 GitHub 的拉取请求（Pull Request），并把拉取请求的地址贴到留言里。这一小节的拉取请求代码，请在档案类专用的代码评审目录下，建一个以你的名字命名的子目录，代码放到你专有的子目录里。比如，我的代码，就放在 record/review/xuelei 的目录下面。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 文字块：怎么编写所见即所得的字符串？

精选留言 (1)

写留言

aoe
2021-11-19

档案类看起来很好用

展开

作者回复: 看起来好用，用起来也好用（哈哈哈）。